The Map Library — A Flexible Group Mechanism for the Intel Paragon XP/S

Samuel A. Fineberg[1]

Report RND-93-015 November 1993

Computer Sciences Corporation
Numerical Aerodynamic Simulation
NASA Ames Research Center, M/S 258-6
Moffett Field, CA 94035-1000
(415)604-4319
`e-mail: fineberg@nas.nasa.gov`

Abstract

"Multidisciplinary" applications are an important class of applications in the area of Computational Aerosciences. In these codes, two or more disciplines are utilized to model a single problem. To support such applications, it is common to use a programming model where an application is divided into several single program multiple data stream (SPMD) codes, each of which solves the equations for a single discipline. These SPMD codes are then bound together to form a single "multidisciplinary" application in which the constituent parts communicate via point-to-point message passing routines. Unfortunately, simple message passing models, like NX, only allow point-to-point and global communication within a single system-defined partition. This makes implementation of multidisciplinary applications quite difficult, if not impossible. In this paper, the design, implementation, and use of the *Map* library, designed at NASA Ames Research Center, is described. The Map library uses a single system group (partition) and allows the user to define arbitrary groups of nodes within the partition. Because these groups are user-defined, there is no system barrier preventing them from communicating. Thus, inter-group communication is possible. In addition, a special version of the NX "global" library is provided to support collective communication within these groups.

1. This work was supported through NASA contract NAS 2-12961.

1.0 Introduction

"Multidisciplinary" applications [BaW93] are an important class of applications in Computational Aerosciences. In these codes, two or more disciplines are utilized to model a single problem. Some disciplines involved in aerospace research include fluid dynamics, thermal analysis, structural analysis, propulsion, etc. To support such applications, it is common to use a programming model where an application is divided into several single program multiple data stream (SPMD) codes, each of which solves the equations for a single discipline. These SPMD codes are then bound together to form a single "multidisciplinary" application in which the constituent parts communicate via point-to-point message passing routines.

Parallel systems, however, typically divide their processors among applications or users by defining a *partition*. Each partition consists of a set of nodes, memory, and typically a single user/application. This type of resource management is also called *space sharing*. A partition is a system managed resource, and partitions are prevented from interacting to provide security. Therefore, partitions are not, ideally, allowed to crash or otherwise interfere with other partitions. Using a partitioned model is useful because it allows the system to provide low overhead security and to have a base scheduling unit. Unfortunately, simple message passing models, like NX, only allow point-to-point and global communication within a single system partition, where all processors in the partition must participate in the operation. This makes implementation of multidisciplinary applications quite difficult, if not impossible. The *Map* library was developed to enable users of the Intel Paragon [Int91] to execute non-homogeneous (e.g., "multidisciplinary") applications, and to provide a more flexible mechanism for defining groups of processors within a single system partition.

1.1 Design Trade-offs

On the iPSC/860, such programs are supported by using the *inter-cube* communication library [Bar91]. Instead of building a flexible group mechanism, this library works with NX's built-in partitioning mechanism. Groups are defined as system partitions, or *subcubes* in this case. The library then uses the mechanism that allows operating system messages to bypass the security barriers between system partitions. Because the iPSC/860's operating system (NX) uses the same communication mechanism as user programs do, there is no performance penalty caused by using system messages for inter-partition communication. Because this method bypasses the security and partitioning mechanism provided by the OS, some features are lost. First, security is compromised, allowing user applications to interfere with one another. Furthermore, a user's application appears to the system as several jobs. This means that the single scheduling unit normally provided by system partitions is lost. As a result, system utilities such as NQS (the batch scheduler) can not schedule jobs that use the inter-cube library.

On the Paragon, the inter-cube library solution is not possible. The Paragon operating system, OSF1/AD, communicates using a mechanism, "NORMA IPC," that is quite different from that used by user applications. This mechanism is unsuitable for user applications because it is difficult to use and has inadequate performance as compared to the Paragon's NX message passing library [TrM93]. The Paragon also provides standard UNIX communication mechanisms such as pipes and sockets that can be used for inter-partition communication. Unfortunately, none of these provide adequate communication performance, and they are far more difficult to use than the standard NX mechanism used within partitions. Further, it is desirable to maintain the system partitioning model for security and scheduling. Thus, an ideal solution would appear to the system as a single partition of the system.

1.2 The Map Library

The Map library uses a single system partition and allows the user to define arbitrary mappings within the partition. Each mapping is a one-to-one map from a user-defined node number to a node number defined by the system. This allows the node numbers to be re-mapped such that different subsets of the system partition may be defined. Each of these subsets will consist of a set of node numbers from 0 to S-1 (S is the number of nodes in the subset) where S<=P (P is the number of nodes in the partition) corresponding to nodes within the system partition. Because these subsets (groups) are user-defined, there is no system barrier preventing them from communicating. Thus, inter-group communication is possible as long as the sending node has the address of the destination node within some map that is defined locally.

While the Map library is intended to be general and can be used to define overlapping or non-overlapping groups for any purpose a user may desire, it is expected that it will generally be used to separate modules within a large program. This is needed for non-SPMD programs where each individual data stream may need to perform point-to-point and collective communication internally as well as point--to-point communication with other data streams. For example, in Figure 1, collective operations (e.g., global synchronization - `gsync`) can occur within either Group A or Group B. In addition, as shown, nodes within the groups can communicate using point-to-point mechanisms (e.g., `csend/crecv`). In addition, nodes in Group A can communicate with nodes in Group B (or vice versa) as long as the nodes have the mapping information for the other group.

Thus, the Map library provides the functionality needed for multidisciplinary applications including flexible group definition, support for multiple binaries, collective communication among arbitrary processor sets, etc. It provides this functionality while still preserving the system partition model, thus facilitating security and scheduling. It does not, however, ensure safety within a user partition. Therefore, there is no mechanism to protect user-defined groups from each other.

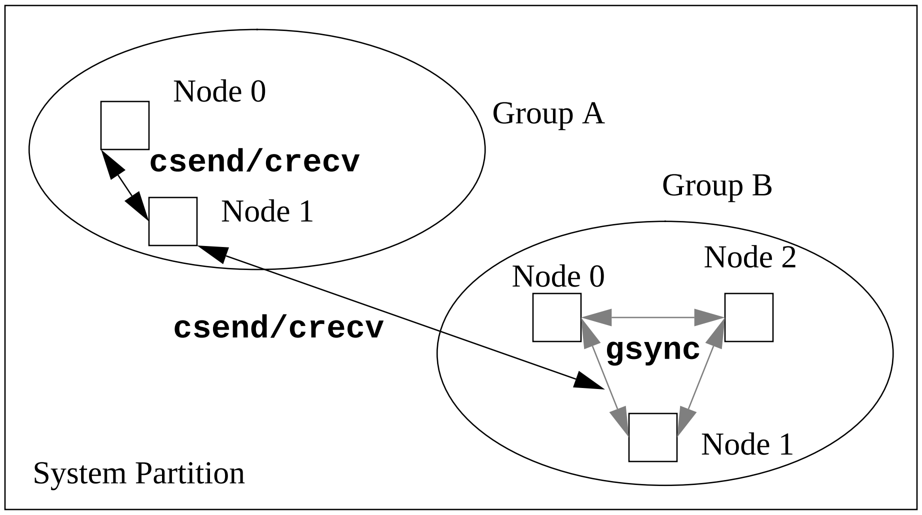

Figure 1: Example of multiple groups within a partition.

2.0 Maps

The primary feature of the Map library is the *map*. A map is a one-to-one function from a logical processor number, or *rank*, to a system-defined processor number. Therefore, one could think of a map as a table containing an ordered list of system-defined processor numbers where a processor's rank within the map is the same as its position within the list. Then, to determine where a message is to be sent, one specifies a map and a rank. This information is then sufficient to determine the system-defined processor number of the destination processor. The system-defined number can then be used for the actual communication. Conversely, because a map is one-to-one, it is possible to determine the rank of a processor within a map as long as one knows the system-defined processor number and the map in which the processor's rank is defined.

A map is a simple one-to-one function and has several possible uses. However, in particular, it is expected that this mechanism will be used for defining groups of processors within system partitions. Therefore, each individual map will represent a group of processors.

The Map library provides mechanisms for defining and manipulating maps. In addition, it provides mechanisms for collective communication between processors that are defined within a map and it provides a mechanism for starting jobs on subsets of the Paragon's processors.

The actual structure of a map is hidden from the user. Instead, maps are referred to via an integer id. In addition, maps are a purely local data structure. Therefore, map 1 on node 8 may represent a completely different mapping than map 1 on node 5. These design decisions make the Map library both fast and extensible. By hiding data structures, the underlying mechanisms can be modified to improve performance and functionality. Forcing maps to be a local data structure means that no map manipulation operations require communication. Thus, they are all fast. The current implementation of the Map library is optimized such that the most common map operations all execute in constant time. More details on this will be presented in Section 5. Unfortunately, this performance is provided at the expense of safety. Therefore, it is up to the programmer to make sure that programs within a partition do not interfere with each other and that the correct map is used for a given collective communication operation. Further, because the debugger does not know about maps, it may be more difficult to debug codes using this library.

3.0 Using the Library

In this section the basic usage of the Map library will be described. It is assumed that the reader of this section is familiar with the NX message passing library on the Paragon. In addition, the reader should be familiar with the Paragon's collective communication (global) library.

While all functions in this section are shown implemented in C, they are also available under FORTRAN. In FORTRAN all of the functions with no return value correspond to FORTRAN subroutines (i.e., to use `map_init()` one should "`CALL MAP_INIT()`" from FORTRAN). Functions that do return a value correspond to FORTRAN functions of type "`INTEGER`."

3.1 Initializing the Library

Before doing any library calls, each node must call

```
map_init()
```

to initialize the basic map data structures and to define an initial map. This initial map, referred to as `map_all`, contains all of the compute nodes in the system partition with their node numbers within the map set to the same value as they would be in the system partition. Therefore, the NX node number equals the node number within `map_all`. Note: `map_all` does not contain the host service node. The host node does not initially belong to any map and its node number (for point--to-point communication) can only be determined with the `myhost()` call.

3.2 Defining Maps

A map is a local data structure managed by the map library. There are two basic methods for creating maps. These functions create a map based on either a list of node numbers or a range of consecutive node numbers. The functions are defined as follows:

```
int map_create_range(int low, int high, char *mapname)
```

creates a map with the name defined in `mapname` consisting of the nodes whose system-defined node number is between `low` and `high`; and

```
int map_create_list(int *list, int num, char *mapname)
```

creates a map consisting of `num` nodes where node number `i` is mapped to system-defined node number `list[i]`. The new map can then be referred to by the integer map number returned by the map creation function.

To free the resources associated with a map, the

```
map_free(int map_num)
```

function may be called. To assist in the defining of maps, a function is provided to allocate lists of nodes. This function:

```
map_allocate_list(int n, int *list)
```

takes as input the number of nodes that one wishes to allocate (`n`) as well as an array with space allocated to hold the list (`list`). The function then attempts to allocate the requested number of nodes from its free list. Each node in a system partition will only be allocated once, and if more nodes are requested than are available, a run-time error will occur. This list can then be used as input to `map_create_list.` Lists can also be accessed from within existing maps.

```
map_print_list(int map_num)
```

prints out the list used to create the map referenced by `map_num` (to standard output). Further, a list can be extracted from an existing map.

```
int map_list(int map_num, int *list)
```

returns the number of elements in the list used to create the map referenced by `map_num`, and a pointer to that list is returned in `list`. This function can also be used to distribute maps among nodes. To send a map to another node, a program must first get the map list (i.e., use the `map_list` function), then the list can be sent to any other nodes in the system and those nodes can create their own copies of the map using the `map_create_list` routine.

3.3 Using Maps

There are two basic map operations. These are `map` and `map_inv.`

```
int map(int map_num, int rank)
```

returns the system node number of the node numbered `rank` within the map referred to by `map_num`.

```
int map_inv(map_num, int num)
```

performs the inverse operation. Therefore, it returns the rank of the node with the system-defined node number `num` within the map referred to by `map_num`.

These functions can then be used for point-to-point communication. For example, to send to node 5 in map 1 the following command could be used:

```
csend(type, buf, n, map(1,5), 0);
```

Further, map_inv can be used to determine who a message came from relative to a known map. Therefore, if a node receives a message and knows that it is from a node in map 5, it can determine the sender as follows:

```
crecv(type, buf, n);
sender = map_inv(5, infonode());
```

To facilitate NX style programs, several standard functions are provided.

```
int map_mynode(int map_num)
```

returns the node number of the calling node relative to the map pointed to by `map_num` or returns -1 if the calling node is not a member of the map.

```
int map_numnodes(int map_num)
```

returns the number of nodes listed in the map referred to by `map_num`. Finally,

```
int map_infonode(int map_num)
```

returns the node number of the sender of the last message received relative to the map pointed to by `map_num`.

3.4 Collective Communication

Collective communication is performed via all of the standard Intel “global” routines (e.g., `gsync`, `gdhigh`, etc.). These exist in the Map library in a modified form with an additional parameter, the integer map number (which defines the map in which the operation should occur). This map number is added to the routines argument list as the last parameter. For example, the map version of `gdlow` is defined as follows:

```
gdlow(double x[], long n, double work[], int map_num)
```

and `gsync` is defined as,

```
gsync(int map_num).
```

The only additions to the collective communication library are two functions for broadcasting. These are needed because it is not desirable to modify the existing `csend` function, and without doing so, the normal NX broadcast mechanism (i.e., sending to a negative processor number) will not work properly within a map. To facilitate different needs, two different broadcasts are provided. Neither of the two broadcasts require the broadcasting node to be a member of the map in which the message is being distributed. The first command,

```
map_bcast(int type, void *buf, int count, int ptype, int map_num)
```

is intended to preserve the semantics of standard NX broadcasting, i.e., the processors receiving the data use the `crecv` command. However, this necessitates the use of a linear time complexity algorithm, so `map_bcast` is quite slow. This function should not be used for most purposes. Instead, the “fast” broadcasting commands that have a log complexity should be used. To broadcast data using the fast algorithm, the sending node executes;

```
map_fbcast_send(int type, void *buf, int count, int ptype,
                int map_num)
```

where `map_num` is the map number used locally to refer to the destination map. Then, the members of the map receiving the data execute;

```
map_fbcast_recv(int type, void *buf, int count, int ptype,
                int map_num)
```

This means that the receiving nodes must have the map information for their map. This may not be the case, particularly when bootstrapping a new map. In those cases the `map_bcast` command must be used. The `map_fbcast` routines have performance similar (i.e., within a few percent, sometimes faster sometimes slower) to standard NX broadcasting.

4.0 Compiling and Loading

4.1 Compiling Hostless Programs

The simplest way to use the library is to use “hostless” programs. The file `map.h` (or `fmap.h` for FORTRAN programs) is “included” in the code and linked with the map library. For example,

```
cc -c -I<ipath> test.c
cc test.o -L<lpath> -o test -lmap -nx
```

where `<ipath>` is the path of the include directory where `map.h` is found and `<lpath>` is the directory where `libmap.a` is found. For FORTRAN, a similar method is used. For example,

```
f77 -c test.f
f77 test.o -L<lpath> -o test -lmap -nx
```

where `<lpath>` is the directory where `libmap.a` is found.

4.2 Compiling Host/Node Programs

The real power and flexibility of the Map library is revealed when multiple binaries are used. For these programs it is necessary to use the host/node model of programming. To compile a “host” or “node” program, simply link with the map library, ensuring that “`map`” appears before “`nx`” in the list of libraries. For example,

```
cc -c -I<ipath> host.c
cc host.o -L<lpath> -o host -lmap -lnx
f77 -c node.f
```

```
f77 node.o -L<lpath> -o node -lmap -lnx
```

The standard NX commands provided by Intel can be used for creating partitions and loading jobs onto nodes. By using the `map_list` and `map_create_list` commands, one can send map information defined in the host program to newly created node processes, using the host program to disseminate information about maps to disparate groups.

4.3 Automatic Map Creation

A facility is provided for automatically creating an entire set of processes and initializing a map containing only the processes that are a member of that set. Therefore, if a multidisciplinary application consists of more than one distinct code, each of these could be started separately from its own binary (i.e., “a.out” file) and the nodes running each particular binary would define a map that contains only those nodes running the same binary. For convenience, a system-defined constant, `map_home` is used to refer to this initial map. Thus, processes created with this method have two pre-defined maps, `map_all`, in which all compute nodes are contained and `map_home`, which contains the nodes whose processes were created at the same time. If this mechanism is not used, node programs will have no information about the group in which they were created.

There are two ways to use this facility. The first method involves the use of a special “main” program. The standard program “main” is replaced with a map version of “main” that takes care of initializing the Map library and creates `map_home`. In C, `main` is renamed `map_main` in the “node” programs. Therefore, a “hello world” node program might appear as below:

```
#include <nx.h>
#include <map.h>

map_main(){
   printf(“Hello from node %d\n”, map_mynode(map_home));
}
```

The node program must then be linked as follows:

```
cc -c -I<ipath> node.c
cc <mpath>/map_main.o node.o -L<lpath> -o node -lmap -lnx
```

where `<mpath>` is the path of `map_main.o`. A node program linked in this manner must *not* call `map_init`. Instead, this is done within the new `main` function. All `argc/argv` parameters will be passed automatically through to the user-defined `map_main` function. For FORTRAN programs, the process is more complicated. First, the “`program xxx`” statement must be changed to a “`subroutine map_fmain`.” Therefore, a FORTRAN “hello world” could appear as follows:

```
   subroutine map_main
```

```
      include ‘map.h’
      print *, ‘Hello from node ‘, map_mynode(map_home)
   end
```

The program is then compiled with f77 and linked with cc as follows:

```
f77 -c -I<ipath> node.f
cc <mpath>/map_fmain.o node.o -L<lpath> -o node -lmap -lf -lm -lnx
```

Note that `cc` is used instead of `f77` because the new “main” program is written in C. In addition, it is vital that the `-lf` and `-lm` be added on the link command line so that all of the expected standard FORTRAN functions will be available.

The other method is to call the `map_init_node` function at the beginning of the node program. This function is called with no parameters and initializes the map data structures including `map_home`. Note that with either of these approaches, a node program should *not* call `map_init.` An example of how to use this function from C follows:

```
#include <nx.h>
#include <map.h>

main(){
   map_init_node();
   printf(“Hello from node %d\n”, map_mynode(map_home));
}
```

Similarly from FORTRAN one simply calls this function. Therefore, the example becomes the following:

```
   program hello
      include ‘map.h’
      call map_init_node()
      print *, ‘Hello from node ‘, map_mynode(map_home)
   end
```

The host program is compiled as before, and the NX environment is established in the standard way (i.e., with `nx_initve`()). However, to load a program onto a set of nodes, one must use the function

```
int map_loadve(int node_list[], int num_nodes, int ptype,
               int pid_list[], char *path, char *argv[],
               char *envp[])
```

This function takes the same arguments as the `nx_loadve` command and essentially performs the same function. However, it also creates a map defined by the list specified in `node_list[]` and returns the map number of this new map. In addition it interacts with the “main” routine or `map_init_node`() function linked to the node program in order to set up `map_home`.

As an example, the following host program will run two instances of the program “hello,” each on half of the allocated nodes (assuming the number of nodes is even, if not, one node will be left out). Note that the number of nodes is specified at run-time using the command line parsing ability of `map_initve()`.

```
#include <stdio.h>
#include <nx.h>
#include <map.h>

void main(int argc, int *argv[]){
   int *list, map1, map2;
   int *pid_list;
/* initialize system partition */
   nx_initve(NULL, 0, NULL, &argc, argv);
   setptype(0);
/* initialize map library */
   map_init();
/* allocate space for node list */
   list = (int *)malloc((map_numnodes(map_all)/2) * sizeof(int));
/* allocate space for pid list */
   pid_list = (int *)malloc((map_numnodes(map_all)/2) *
               sizeof(int));
/* get a list containing 1/2 of the node */
   map_allocate_list(map_numnodes(map_all)/2, list);
   printf(“Running 1st hello\n”);
/* start “hello” on the listed nodes */
   map_loadve(list, map_numnodes(map_all)/2, 0, pid_list, “hello”,
               NULL, NULL);
/* allocate another list */
   list = (int *)malloc((map_numnodes(map_all)/2) * sizeof(int));
   map_allocate_list(map_numnodes(map_all)/2, list);
   printf(“Running 2nd hello\n”);
/* start “hello” on the rest of the nodes */
   map_loadve(list, map_numnodes(map_all)/2, 0, pid_list, “hello”,
               NULL, NULL);
   printf(“Waiting\n”);
/* wait for everything to complete */
   nx_waitall();
}
```

5.0 Implementation

This briefly describes some of the internal implementation features of the current version of the Map library (Version 1.0 beta as of 9/93). All of these details are subject to change in later versions of the library. The primary goal of this initial implementation of the library is performance. In particular, several trade-offs are made where memory is sacrificed in favor of some performance enhancement feature. In addition, several hard-coded limits are imposed. In particular, the current version of the map library only allows up to 256 maps. This restriction on the number of maps will likely be removed in later versions of the library.

5.1 Map Data Structure Implementation

Maps are represented internally by the following C data structure:

```
static struct mapping {
   int *list;
   int *inv_list;
   int mapsize;
   char *mapname;
   } maplist[NUM_MAPS];
```

where `NUM_MAPS` is predefined to be 256. In this data structure, `list` represents the linear list of physical node numbers used when creating a map (i.e., `list[i]` is the system node number of the node with rank `i` in the map). `inv_list` is a linear list with `numnodes()+1` elements (i.e., one entry for each node in a system partition plus an extra for the host node) containing either the rank of the corresponding system node within the map or -1 (the value of the constant `MAP_UNDEF`) if the node is not a member of the map. This mechanism wastes memory, especially if the system partition size is large, but allows constant time inverse mappings. This inverse mapping function turns out to be used frequently, so it is important that it be fast. `mapsize` is simply the number of nodes within a map and `mapname` is a character name for a map. Note that currently the map name, `mapname`, is unused. However, this parameter is added so that maps can be referred to by a name instead of just a local number and to make it possible to develop a map-server in the future.

The map list is managed using a linear “free” list and map structures can be removed from and added back to the free list by “creating” or “freeing” a map. When a map is created, the map data structure is allocated from the free list. Then, the map’s list is created either by setting the value of the list pointer to a user-defined list or by creating a list from information provided (e.g., from a range of node numbers as in `map_create_range()`). Then, the “create” function allocates memory for and defines the inverse mapping list, and finally sets the size and name of the list appropriately.

This definition of a map allows the `map` and `map_inv` functions to be very fast. `map()` is simply implemented as follows:

```
int map(int map_num, int rank){
   return(maplist[map_num].list[rank]);
}
```

and `map_inv()` is equally simple:

```
int map_inv(int map_num, int num){
   if (map_num == MAP_ALL) return(num);
   return(maplist[map_num].inv_list[num]);
}
```

Because these are likely to be the two limiting cases for performance (i.e., one of these is called at least once for every map library function), it is important that they be as fast as possible. In particular, if the inverse list was not provided, the `map_-inv()` function would be O(N) complexity (this is because a binary search will not work without sorting the map first). Instead, this mechanism returns an inverse mapping in constant time with a cost of only 4 bytes of memory per node in the system-defined partition.

5.2 Collective Communication Implementation

The global library included in the current implementation of the Map library consists of modified source code for `libnx.a`, as provided by Intel with Paragon OS Transmittal 10. All of the "global" calls in the NX library are implemented using standard send and receive calls. Therefore, the only modifications needed are:

1. An extra parameter is added to each function for the map id.
2. The node number in every send and receive call is replaced by the appropriate call to `map()`.
3. Calls to helper functions such as `mynode()`, `infonode()`, etc. are replaced by the appropriate map versions.

The only other problem with this mechanism is for broadcasting. Normal NX broadcasting is accomplished through the use of a negative node number when calling `csend()`. Modifying this mechanism is difficult, and might have some unwanted side effects (i.e., reduction in performance or increase in complexity for normal sends and receives), so separate map broadcasting mechanisms are provided instead. When broadcasting within maps, there are several difficulties. First, it may be desirable to be able to broadcast to a map one is not a member of (i.e., a host may want to broadcast to a set of nodes). Further, nodes may not know what map is being broadcasted to when they receive a broadcast message. For example, if a host wants to create a new group of nodes, it may simply broadcast a map list to the members of the new group. However, because the map has not been created yet, the nodes do not yet know the members of the group to which the message is being broadcasted. Finally, for backward compatibility, it is desirable to have a broadcasting mechanism that could be received by a standard `crecv()` call as is the case for NX style broadcasting.

These issues led to the development of two broadcasting mechanisms. The first, `map_fbcast_send()` and `map_fbcast_recv()` are fast mechanisms for broadcasting when all nodes have the mapping information for the group being broadcasted to. This function first sends the data to node 0 of the map (thus, allowing non-members to broadcast), then uses a recursive doubling algorithm to distribute the data. Note that while recursive doubling may not be the optimal algorithm for a mesh, it minimizes the number of send/receive calls, and since there is no guarantee that a map will be a mesh, this is the best one can do. The performance of this function is O($\log_2 N$) and has been measured to be approximately the same as NX broadcasting (faster in some cases, slower in others).

The second broadcasting mechanism, `map_bcast()`, simply sends data items one at a time to each node in a group. In the fast broadcast mechanism, a communication tree is formed. Therefore, the receiving nodes may have to send one or more messages to other members of the receiving group. This means that a special receive function is needed and the receiving nodes must know about the other members of their group. However, with `map_bcast()` no communication is needed after receiving the data, so the receiving nodes can simply use the standard `crecv()` function. Further, the receiving nodes do not need to know the map defining the receiving group. Unfortunately, this mechanism is quite slow, particularly for large machine sizes (its complexity grows O(N)).

6.0 Conclusion

In this paper the Map library for the Paragon XP/S has been described. The Map library provides the support necessary for multidisciplinary applications. It does so by implementing a flexible group mechanism for programs on the Paragon as well as full support for collective communication and multiple program binaries. This library was designed to be fast and extensible. An initial implementation is currently available for the Paragon and similar libraries are in development for other Parallel machines, including the Thinking Machines CM-5.

7.0 Acknowledgments

I would like to acknowledge the Message Passing Interface Forum and in particular the Contexts subcommittee from which I derived many of the ideas for this library. I would also like to acknowledge Eric Barszcz and Sisira Weeratunga for providing information on the requirements for the library and for testing it. Finally, I wish to thank Intel's Supercomputer Systems Division for providing up-to-date source code for the NX message passing library. Without this support, my job would have been much more difficult.

8.0 References

[Bar91] E. Barszcz, *Intercube Communication for the iPSC/860*, NASA Ames Research Center, Report Number RNR-91-030, October 1991.

[BaW93] E. Barszcz, S. Weeratunga, and E. Pramono, *A Model for Executing Multidisciplinary and Multizonal Programs*, NASA Ames Research Center, Report Number RNR-93-009, March 1993.

[Int91] Intel, *Paragon XP/S Product Overview, Intel Corporation,* Supercomputer Systems Division, Beaverton, OR, 1991

[TrM93] B. Traversat, D. McNab, B. Nitzberg, and S. Fineberg, "Evaluation metrics for the Intel Paragon XP/S-15," *Intel Supercomputer Users' Group 1993 Annual North America User's Conference*, St. Louis, MO, October 1993, to appear.

RND TECHNICAL REPORT
Title: The Map Library - A Flexible Group Mechanism for the Intel Paragon XP/S
Author(s): Samuel A. Fineberg
Reviewers: “I have carefully and thoroughly reviewed this technical report. I have worked with the author(s) to ensure clarity of presentation and technical accuracy. I take personal responsibility for the quality of this document.” Signed: ______________________________ Name: ______________________________ Signed: ______________________________ Name: ______________________________
Branch Chief: Approved: ______________________________
Date & TR Number: